

NO-A188 613

MULTIRPC: A PARALLEL REMOTE PROCEDURE CALL MECHANISM
(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER
SCIENCE M SATYANARAYANAN ET AL DEC 87

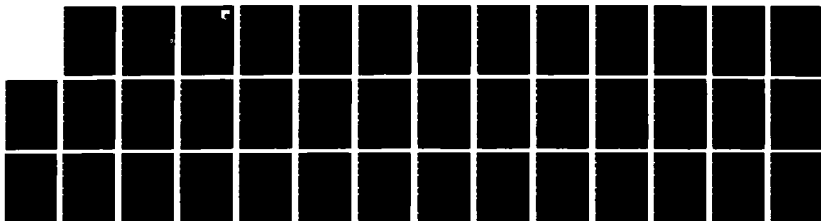
1/1

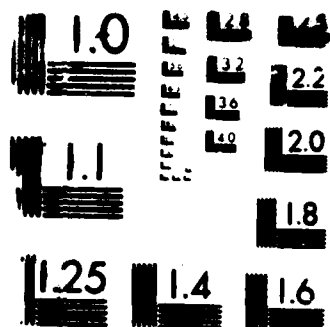
UNCLASSIFIED

CNU-CS-87-136-A AFWAL-TR-87-1171

F/G 12/7

ML





• COPY RESOLUTION TEST CHART

PHOTOGRAPH THIS SHEET

AD-A188 613

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

AFWAL-TR-87-1171

DOCUMENT IDENTIFICATION

This document has been approved
for public release and sale; its
distribution is unlimited.

DISTRIBUTION STATEMENT

ACCESSION FOR

NTIS GRA&I

DTIC TAB

UNANNOUNCED

JUSTIFICATION



BY

DISTRIBUTION

AVAILABILITY CODES

DIST

AVAIL AND/OR SPECIAL

A-1

DISTRIBUTION STAMP



DTIC
ELECTE
FEB 09 1988
S D
A E

DATE ACCESSIONED

DATE RETURNED

88 2 05 097

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NO.

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

AD-A188 613

AFWAL-TR-87-1171

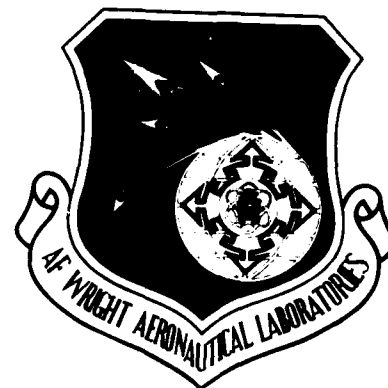
MultirPC: A PARALLEL REMOTE PROCEDURE CALL MECHANISM

M. Satyanarayanan
E.H. Siegel

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim



Approved for Public Release; Distribution is Unlimited

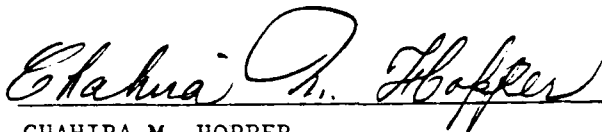
AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

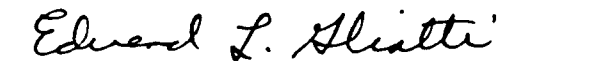


CHAHIRA M. HOPPER
Project Engineer



RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER



EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-CS-87-136-A			5. MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-87-1171		
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University		6b. OFFICE SYMBOL (if applicable)		7a. NAME OF MONITORING ORGANIZATION Air Force Wright Aeronautical Laboratories AFWAL/AAAT-3	
6c. ADDRESS (City, State, and ZIP Code) Computer Science Dept Pittsburgh PA 15213-3890			7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6543		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (if applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-84-K-1520	
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 61101E	PROJECT NO. 4976	TASK NO. 00
11. TITLE (Include Security Classification) MultiRPC: A Parallel Remote Procedure Call Mechanism					
12. PERSONAL AUTHOR(S) M. Satyanarayanan; E. H. Siegel					
13a. TYPE OF REPORT Interim		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1987 December	
15. PAGE COUNT 38					
16. SUPPLEMENTARY NOTATION This is a revised version of report CMU-CS-86-139.					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>MultiRPC is an invocation mechanism that enables a client to access multiple servers in a single remote procedure call. Parallelism is obtained from concurrency of processing on servers and from the overlapping of retransmissions and timeouts. Each of the parallel calls retains the semantics and functionality of the underlying remote procedure call mechanism. These include secure, authenticated communication and the use of application-specific side effects. The underlying communication medium does not have to support multicast or broadcast transmission. In this paper we describe the design and evolution of MultiRPC, focusing on the issues of runtime efficiency, versatility, and ease of use. We derive an analytic model of the system and present experimental results that validate this model. We also present our observations on using MultiRPC to contact up to 100 servers in parallel.</p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Chahira M. Hopper			22b. TELEPHONE (Include Area Code) (513) 255-7865		22c. OFFICE SYMBOL AFWAL/AAAT-3

Table of Contents

1. Introduction	1
2. Overview of RPC2	1
3. Motivation	3
4. Design Considerations	4
5. Design and Implementation	4
5.1. Overall Structure	4
5.2. Handling Failures	5
5.3. Evolution	6
6. Performance	7
6.1. Analytic Model	8
6.2. Experimental Results	10
6.2.1. Experimental Environment	10
6.2.2. Validation	11
6.2.3. Large Scale Effects	11
7. Related Work	12
8. Conclusion	13
Acknowledgements	13
I. An Example	14
II. Network Topology	19
III. Tables	20
IV. Graphs	25

1. Introduction

RPC2 is a remote procedure call mechanism that has been used extensively in the Andrew distributed computing environment at Carnegie Mellon University [12]. A detailed description of *RPC2* may be found elsewhere [16, 15]. *MultiRPC* is an extension to *RPC2* that enables a client to perform remote invocations of multiple servers while retaining the reliability characteristics of remote procedure calls. In this paper we describe *MultiRPC* and show how we have made it fast, versatile and simple to use.

Section 2 presents an overview of *RPC2*. Section 3 explains why we extended *RPC2* with a parallel invocation mechanism. The considerations that influenced the design of *MultiRPC* are put forth in Section 4. Section 5 describes the design and implementation of *MultiRPC*, explores some of the subtle consequences of our original design decisions, and motivates the resulting modifications. Section 6 discusses the experimental evaluation of the system. An analytic model is derived, and validated by comparing its predictions to the results of controlled experiments. Section 7 relates this work to other efforts relating to parallelism in network communication. Section 8 concludes the paper with an overview of work in progress.

2. Overview of *RPC2*

RPC2 consists of two relatively independent components: a Unix-based runtime library written in C, and a stub generator, *RP2Gen*. The runtime system is self-contained and is usable in the absence of *RP2Gen*. The code in the stubs generated by *RP2Gen* is, however, specific to *RPC2*.

A *subsystem* is a set of related remote procedure calls that make up a remote interface. *RP2Gen* takes a description of a subsystem and automatically generates code to marshall and unmarshall parameters in the client and server stubs. It thus performs a function similar to *Lupine* in the Xerox RPC mechanism [1] and *Matchmaker* in Accent IPC [10].

The *RPC2* runtime system is fully integrated with a Lightweight Process mechanism (*LWP*) [13] that supports multiple nonpreemptive threads of control within a single Unix process. When a remote procedure is invoked, the calling *LWP* is suspended until the call is complete. Other *LWPs* in the same Unix process are, however, still runnable. The *LWP* package allows independent threads of control to share virtual memory, a feature that is not present in standard Unix. Both *RPC2* and the *LWP* package are entirely outside the Unix kernel and have been ported to multiple machine types.

The low-level packet¹ transport mechanism is a separable component of *RPC2*. The only primitives required of it are the ability to send and receive datagrams. At present, *RPC2* runs on the DARPA IP/UDP protocol [6, 7].

RPC2 is based on logical *connections*. The rationale for choosing a connection-based rather than connectionless protocol is presented in the design document [16]. For the purposes of this paper the following facts about *RPC2* connections are relevant:

1. A connection is created when a client invokes the *BIND* primitive and is destroyed by the *UNBIND* primitive. The cost of a *BIND* is comparable to the cost of a normal RPC.
2. One can view *BIND* as a special RPC that is common to all subsystems. In fact, a server is notified of the creation of a new connection in exactly the same way it is notified of an RPC on an existing connection.

¹Throughout this paper we use the term "packet" to mean a logical packet. In some networks a packet may be physically transmitted as multiple fragments. Such fragmentation is transparent to *RPC2*.

3. Connections use little storage. Typically a connection requires a hundred bytes at each of the client and server ends. No other resources are used by a connection.
4. Within each Unix process, an RPC2 connection is identified by a unique *handle*. Handles are never reused during the life of a process.
5. At any given time a Unix process can have at most 64K active connections. This is about two orders of magnitude larger than the number of connections in use in the most heavily loaded Andrew servers.

Although one speaks of "clients" and "servers", it should be noted that the mechanism is completely symmetric. A server can be a client to many other servers, and a client may be the server to many other clients. On a given connection, however, the roles of the peers are fixed.

Besides BIND and UNBIND, the most important runtime primitive on the client side is MAKERPC. This call sends a request packet to a server and then waits for a reply. Reliable delivery is guaranteed by a retransmission protocol built on top of the datagram transport mechanism. Calls may take an arbitrary length of time; in response to client retries the server sends keep-alive packets (*BUSY* packets) to indicate that it is still alive and connected to the network. On the server side, the basic primitives are GETREQUEST, which blocks until a request is received and SENDRESPONSE, which sends out a reply packet. RPC2 provides *exactly-once* semantics in the absence of site and hard network failures, and *at-most-once* semantics otherwise [17].

A unique aspect of RPC2 is its support of arbitrary *side effects* on RPC calls. The side effect mechanism allows application-specific protocols to be integrated with the base RPC2 code. Side effects and a number of other RPC2 features are discussed in the design document. Tables 2-1, 2-2, and 2-3 summarise the RPC2 primitives relevant to this paper.

Primitive	Description
BIND	Create a new connection
MAKERPC	Make a remote procedure call
MULTIRPC	Make a collection of remote procedure calls

Table 2-1: Client Primitives

Primitive	Description
EXPORT	Indicate willingness to accept calls for a subsystem
DEEXPORT	Stop accepting new connections for one or all subsystems
GETREQUEST	Wait for an RPC request or a new connection
ENABLE	Allow servicing of requests on a new connection
SENDRESPONSE	Respond to a request from a client
INITSIDEFFECT	Initiate side effect
CHECKSIDEFFECT	Check progress of side effect

Table 2-2: Server Primitives

Primitive	Description
INIT	Perform runtime system initialisation
UNBIND	Terminate a connection by client or server
ALLOCBUFFER	Allocate a packet buffer
FREEDBUFFER	Free a packet buffer

Table 2-3: Miscellaneous Primitives

3. Motivation

The principles underlying MultiRPC arose as a solution to a specific problem in Andrew. In the Andrew file system [14], workstations fetch files from servers and cache them on their local disks. In order to maintain the consistency of the caches, servers maintain *callback* state about the files cached by workstations. A callback on a file is essentially a commitment by a server to a workstation that it will notify the latter of any change to the file. This guarantee maintains consistency while allowing workstations to use cached data without contacting the server on each access. Before a file may be modified on the server, every workstation that has a callback on the file must be notified. Since the system is ultimately expected to encompass over 5000 workstations, an update to a popular file may involve a callback RPC to hundreds or thousands of workstations. The problem is exacerbated by the fact that a callback RPC to a dead or unreachable workstation must time out before the connection is declared broken and the next workstation tried. Each such workstation would cause a delay of many seconds, rather than the few tens of milliseconds typical of RPC roundtrip times for simple requests. Given these observations, we felt that the potential delay in updating widely-cached files would be unacceptable if we were restricted to using simple RPC calls iteratively.

A simple broadcast of callback information is not feasible. With broadcast, every time a file is changed anywhere in the system every workstation would have to process a callback packet and determine if the packet were relevant to that workstation. Using multicast to narrow the set of workstations contacted is also impractical, because each file would then potentially have to correspond to a distinct multicast address. Since workstations flush and replace cache entries frequently, the membership of multicast groups would be highly dynamic and difficult to maintain in a consistent manner.

Besides these considerations, the use of broadcast or multicast does not provide servers with confirmation that individual workstations have indeed received the callback information. Such confirmation is implicit in the reliable delivery semantics of RPC. It became clear to us that we needed a mechanism that retained strict RPC semantics while overlapping the computation and communication overheads at each of the destinations. This is the essence of MultiRPC.

MultiRPC has applications in other contexts too. Replication algorithms such as quorum consensus [9] require multiple network sites to be contacted in order to perform an operation. The request to each site is usually the same, although the returned information may be different. MultiRPC could be used to considerably enhance the performance of such algorithms. The performance of some relatively simple but frequent operations in large distributed systems may also be improved by MultiRPC. Consider, for example, the contacting of a name or time server. If more than one such server is available, it may be reasonable to use MultiRPC to contact many of them, wait for the earliest reply and abandon all further replies.

4. Design Considerations

The primary consideration in the design of MultiRPC was that it be inexpensive. We did not want normal RPC calls to be slowed down because of MultiRPC. Although one-to-many RPC calls constituted a very important special case, we expected simple, one-to-one RPC calls to be preponderant. A related, but distinct, concern was the increase in program size resulting from MultiRPC. Since virtual memory usage in our workstations was already high, we wished to keep MultiRPC small.

Another influence on our design was the desire to decouple the design of subsystems from considerations relating to MultiRPC. We did not want to require any changes to clients who used only RPC2, or to servers. Our view was that only clients who wished to access multiple sites in parallel should have to know about MultiRPC.

Since we insisted on allowing simple RPC and MultiRPC calls in any order on any combination of connections, MultiRPC had to be completely orthogonal to normal RPC2 features. The delivery semantics, failure detection, support for multiple security levels and the ability to use side effects all had to be retained when making a MultiRPC call.

A number of the scenarios in which we envisaged MultiRPC being used required replies to be processed by the client as they arrived rather than being batched. Since the exact nature of such processing was application dependent it had to be performed by a client-specified procedure. In addition, we felt that it was important for a client to be able to abort the MultiRPC call either after examining any reply or after a specified amount of time had elapsed since the start of the MultiRPC call.

Finally, we wanted MultiRPC to be simple to use. We have been successful in this even though the syntax of a MultiRPC call is different from the syntax of a simple RPC call, the latter being similar to a local procedure call. We have had to violate this syntax for two reasons: to allow clients to specify an arbitrary reply-handling procedure in a MultiRPC call, and to avoid expanding code size by generating a MultiRPC stub for every call in a subsystem.

5. Design and Implementation

In this section we first present the design of MultiRPC. We discuss certain reliability and performance problems revealed by a prototype implementation and then describe the refinements made to alleviate these deficiencies.

5.1. Overall Structure

Support for MultiRPC is present at both the runtime level and the language level, reflecting the organisation of RPC2. The runtime interface can be used independent of the language interface, but not vice versa.

Runtime support is provided by the routine `MULTIRPC` that takes a request packet, a list of connections and a *client handler* routine as input, and blocks until all responses have been received or until the call is explicitly terminated by the client handler. The packet which is transmitted to a server is identical to a packet generated by an RPC2 call. A server is not even *aware* that it is participating in a MultiRPC call.

MultiRPC provides the same correctness guarantees as RPC2, except when the client terminates a call prematurely. In this case, a success return code indicates that no connection failures were detected prior to the point of termination. However, undetected server failures may have occurred after termination.

Language support for MultiRPC is provided by the routines MAKEMULTI and UNPACKMULTI. These routines interpret templates called *argument descriptor structures (ARGs)* generated by RP2Gen to perform the packing and unpacking of parameters in request and reply packets. The decision to interpret ARGs at runtime rather than to use precompiled stubs as in RPC2 was motivated by storage size considerations. The slight additional processing cost of parameter interpretation is outweighed by the savings in the code size. A consequence of this is that the syntax of a MultiRPC call no longer resembles invocation of a local procedure. Each component of a MultiRPC call does, however, retain the semantics of an equivalent RPC2 call.

Appendix I describes the external interface of MultiRPC using an example. It presents a simple RPC2 subsystem in Figure I-1 and shows typical client and server code written by a user for non-MultiRPC calls in Figures I-3 and I-4. RP2Gen uses the subsystem definition to generate a header file (Figure I-2) as well as client and server stub files (not shown).

Figure I-5 shows how the user has to modify the client code to use MultiRPC. There are only two significant changes: the direct call to the client stub has to be replaced by an indirect call via MAKEMULTI, and a client handler routine has to be provided to process replies.

The ARGs used by MAKEMULTI are defined by RP2Gen in the header file. Each routine in a subsystem has an associated array of ARGs, with the type, usage and size of each parameter being specified by one array element. Structures are described by an array of ARGs, one ARG per field. Nested structures are described by correspondingly nested ARGs. At runtime, MAKEMULTI traverses the ARG array and actual parameter list in step.

The client handler is activated exactly once for each connection specified in the MultiRPC call. Each activation corresponds to the receipt of a reply or to detection of a permanent failure on that connection. The handler enables these events to be processed as soon as they occur. Its return code indicates whether the MultiRPC call should be continued or terminated.

The internal routine SENDPACKETSRELIABLY is the heart of the MultiRPC retransmission, failure detection and result gathering mechanism. It performs an initial transmission of requests on all relevant connections and then awaits replies or timeouts. Each timeout on a connection causes a retransmission of the request to the corresponding server. On a reply, appropriate side effect processing is performed and then UNPACKMULTI is invoked. Client-specified timeouts are handled in this routine.

5.2. Handling Failures

Two factors complicate the semantics of failures in MultiRPC. First, since multiple connections are involved, how does one treat failures on an individual connection? Should the entire call be declared a failure and aborted at that point? In our design this decision is delegated to the client handler. This routine is called on each failure and the return code from it specifies whether the call should be terminated. This allows applications to use a variety of strategies, such as termination on a single failure or termination beyond a threshold of failures. If an error such as an attempt to use a dead connection is detected during initial processing of a MultiRPC request, packet transmission is suppressed on that connection.

The second source of complexity arises from the fact that the client handler can terminate a MultiRPC call before all replies are received. What is the state of the connections on which replies have not been received? Should these connections be monitored for failure after the call is terminated? How are the outstanding replies dealt with if they do arrive eventually? Our strategy is to pretend that a response has actually been received on each of the outstanding connections. After termination of a MultiRPC call MultiRPC increments

the sequence number and resets the state on each such connection. Responses that do eventually arrive are ignored, and new failures will not be detected until the next MultiRPC or simple RPC2 call.

This ability to terminate a MultiRPC call prematurely interacts with an orthogonal aspect of RPC2 to produce a race condition. Originally, the RPC2 protocol required the client to send an acknowledgement to the server when a reply was received. The server would retry the reply until it received the acknowledgement or until it timed out. Suppose a client were to terminate a MultiRPC call prematurely and then immediately make another MultiRPC call. Then deadlock could arise on each connection on which a reply was outstanding when the first call was terminated. The retried replies by the server on that connection would be ignored by the client. Similarly, the server would ignore the new request from the client. Only a server or client timeout could end the deadlock. This problem would be compounded if the client terminated the second call prematurely, and then continued with further MultiRPC calls. The client could continue indefinitely in this mode without realising that the connection was functionally dead.

Our solution to this problem is to send an explicit negative acknowledgement if a packet with a sequence number higher than expected is received. This enables both the client and the server to immediately detect the failure mode described above. Because the RPC2 protocol no longer requires replies to be acknowledged, this fix is now superfluous. However, we retain it to allow prompt identification of connections that have been marked unusable by a server for other reasons such as side-effect failures.

Another possible failure mode relates to the client handler routine. During an excessively long computation in this routine, the internal buffers in Unix will be filled with incoming replies and further replies will be lost. This has the effect of increasing retransmissions and hence degrading performance. In addition, logical errors can arise if the client handler is not reentrant but yields control. This can happen, for instance, if the client handler makes an RPC during its processing. Writing a client handler is thus, in many ways, similar to writing an interrupt handler in an operating system.

5.3. Evolution

Experience with an initial prototype of MultiRPC led us to make a number of changes pertaining to function and performance. The changes relating to function have been mentioned in Section 5.2. In this section we describe the changes that we made to improve the performance of MultiRPC.

Early trials of MultiRPC showed a surprisingly large number of retried packets, even when the number of servers being contacted was relatively small. Careful examination of the code showed that most of these packets were not being lost, but were being discarded after receipt. It turned out that the low-level RPC2 code first checked for timed-out events and then checked for packet arrivals. For a MultiRPC call to many sites, the total time to transmit all requests exceeded the first retransmission interval. Replies from the first few servers were discarded because they corresponded to events that had timed out. To fix this problem we now time out events only after receiving all packets that have arrived.

Another change was made to the same piece of low-level code to reduce the number of LWP context switches. Rather than yield control on each received packet, the code now yields control only after all available packets have been received. In MultiRPC this reduces context switches because all these packets are destined for the same client LWP. This is in contrast to the situation in simple RPC2 calls where the semantics of RPC guarantees that a client LWP can be waiting for at most one packet.

A third change addressed the fact that Unix provides only a limited amount of buffering in the kernel for incoming packets. For a sufficiently large number of servers in a MultiRPC call, enough replies could arrive

While requests were still being sent that the kernel buffer could overflow. To reduce the likelihood of this happening, we now send control packets only when sending packets allows the user-level RPC2 code to receive replies and empty the kernel buffer.

Memory allocations in the user where we now made mistakes. A number of virtual data structures are associated with each connection for the duration of a MultiRPC call. Although the number of connections in a MultiRPC call is known only at runtime, our prototyping code allocated these data structures for simplicity. As is to be expected in static allocation schemes, this limited the maximum number of sites that could participate in a call and at the same time required a substantial amount of memory to be permanently allocated. We now allocate these structures as needed, but never deallocate them. This approach, resulting from the limitations of static allocation, is a memory leak, but since every MultiRPC call RPC2 execution is necessary only when a MultiRPC call starts more sites than have ever been attached to the current process.

The extension to MultiRPC required a number of hidden assumptions in the underlying RPC2 code. In particular, there were at least two code segments whose processing time was quadratic in the number of outstanding events. As discussed earlier in this section, this was never a problem for simple RPC2 calls since there could only be one outstanding event per LWP, and since most applications used only a few LWPs. These pieces of code have now been modified to be linear, rather than quadratic.

A final factor that affected MultiRPC performance significantly was an unexpected interaction with the underlying RPC2 reliable transmission protocol. The original protocol required a client to acknowledge the reply from a server. In most cases the acknowledgement was piggy-backed on the next request from the client, thus minimising the total number of packets exchanged. Unfortunately this optimisation did not work well in MultiRPC when many servers were involved. The total time to send out all the requests was then large enough that the first servers to respond would timeout and retransmit their reply. This increased both the total number of packets exchanged as well as the memory and processor utilisation at the client to queue and process these packets. The additional processing further slowed the client and caused it to lose new replies, thus leading to an unstable mode of operation. For this reason, as well as the failure mode described in Section 5.2 and other reasons independent of MultiRPC, we have changed the reliable transmission protocol to no longer acknowledge replies.

The current implementation of MultiRPC incorporates all the modifications described in this section and a number of other minor changes. The performance measurements described in Section 6 were obtained with this implementation.

6. Performance

The performance measure that best characterises MultiRPC is the ratio of the elapsed time for using RPC2 iteratively (r), to the elapsed time for using MultiRPC (m). This ratio (r/m), as a function of the number of sites contacted (n), is the speedup realised by a MultiRPC implementation. Although linear speedup is clearly desirable, MultiRPC can be valuable even with modest speedup. In the application which motivated MultiRPC, for instance, rapid failure detection was of much greater concern than speedup of processing. This benefit of using MultiRPC exists even if there is no speedup of processing.

In this section we assess the speedup of MultiRPC in three steps. We first present an analytic model in Section 6.1, validate this model using data from controlled experiments in Section 6.2.2, and then present, in Section 6.2.3, data from large-scale experiments where the assumptions behind our model are violated. The raw data for the measurements and the analytic model predictions are found in Appendix III, and are

presented graphically in Appendix IV.

6.1. Analytic Model

Our goal in this section is to derive an analytic model that can predict the behaviour of MultiRPC. Although the simplifying assumptions we make may not strictly hold in practise, they are acceptable for the level of accuracy we are trying to achieve.

The most important assumption deals with network topology and latency. In most distributed systems, the actual transit time on the network is a small fraction of the processing time spent in sending and receiving a packet. Routers or other interconnecting elements on a multi-segment network can, however, increase latency considerably. For the purposes of our model, we assume that the client and all the servers are on a single-segment network that has negligible latency.

A second assumption relates to mutual interference and loss of packets. Although MultiRPC is built on unreliable datagrams, the actual probability of packet loss is quite low, typically below 1 percent. However, as more servers are contacted, the probability of packet loss increases because of limited buffering capability at the client. Even if packets are not lost, race conditions between the client and the servers can cause packet retransmissions. We ignore all these complications and assume that there are no lost or retried packets during a MultiRPC call.

Finally, we assume that each server takes a constant amount of time to service a request and that this time is uniform across all servers. This assumption is valid to a first approximation even though the specific nature of the request, the presence of other processing activity at the servers, and slight differences in hardware performance can result in nonuniform service times.

A MultiRPC call can be decomposed into the following components:

<i>pack</i>	Packing of arguments by client.
<i>cloh</i>	Protocol and kernel processing by client to send request.
<i>servoh</i>	Protocol and kernel processing by server to receive request and send reply.
<i>clproc</i>	Protocol and kernel processing by client to receive reply
<i>unpack</i>	Unpacking of arguments and processing in client handler.

In terms of the MultiRPC implementation described in Section 5.1, *pack* is the time taken by the routine `MAKEMULTI`, *cloh* corresponds to the time in `MULTIRPC` and the initial part of `SENDPACKETSRELIABLY`, *clproc* corresponds to the remainder of `SENDPACKETSRELIABLY`, and *unpack* is the time taken by `UNPACKMULTI` and a call to a null client handler routine. Since MultiRPC and simple RPC2 calls are indistinguishable at the servers, *servoh* is the same for both iterative RPC2 calls and MultiRPC. This is the total time taken to receive a request and to send a reply, assuming zero processing time. We include a separate term *comptime* to account for the application processing at a server.

The *pack* component is performed only once, regardless of the number of servers being contacted. The *servoh* and *comptime* components overlap at the servers. All the other components have to be performed once for each server. In terms of these components, the total time *m* for a MultiRPC call to *n* sites can be expressed as

$$m = \text{pack} + (n \times \text{cloh}) + (\text{servoh} + \text{comptime}) + (n \times \text{clproc}) + (n \times \text{unpack})$$

Unfortunately this expression contains an oversimplification that affects the model predictions significantly. Suppose *waittime* is the elapsed time between the sending of the last request and the receipt of the first reply by the client. For a single server, *waittime* will be the sum of *servoh* and *comptime*. For a large enough

number of servers, however, the reply from the first server may be available before the last request is sent out; in this case *waittime* is zero.

Assuming that the time to send a packet is *sendtime*, the expression for *m* can be refined as follows:

$$\begin{aligned} \text{waittime} &= (\text{servoh} + \text{comptime}) - ((n-1) \times \text{sendtime}) \\ \text{if } (\text{waittime} < 0) \text{ then } \text{waittime} &= 0 \\ m &= \text{pack} + (n \times \text{cloh}) + \text{waittime} + (n \times \text{clproc}) + (n \times \text{unpack}) \end{aligned}$$

If a null RPC2 call takes *rpctime*, the total time, *r*, to contact *n* servers using iterative RPC2 calls is given by

$$r = n \times (\text{rpctime} + \text{comptime}).$$

The times for the individual components in our implementation were obtained by actual measurement and are presented in Table III-2. Using these values in the expressions derived above we can calculate the quantities *r*, *m* and *r/m* for server computation times of 10, 20 and 50 milliseconds. These predicted values are presented in Table III-6 and shown graphically in Figure IV-4.

Most systems with parallelism initially exhibit linear speedup, then show sublinear speedup, and finally saturate. Figure IV-4 shows that MultiRPC conforms to this expected behaviour. However, two detailed observations are apparent from this graph. First, saturation occurs at surprisingly low levels of speedup. Second, the level at which speedup saturates and the number of servers at which saturation sets in are both dependent on the server computation time.

The server computation time is central to MultiRPC because most of the parallelism comes from overlap of server computations. The sending of requests and processing of replies at the client are done sequentially. The realisable speedup depends on how long these operations take in comparison to the time spent at the server.

We can quantify this reasoning in the following way. Our measurements show that the dominant components of MultiRPC are the time to send a request and the time to receive a reply. Suppose the sum of these quantities, called *sysstime*, is identical in MultiRPC and RPC2. Further, let the total time spent at a server (equal to the sum of *servoh* and *comptime*) be *servtime*. Then the MultiRPC and RPC2 call times to *n* servers can be crudely approximated as follows:

$$\begin{aligned} m &= (n \times \text{sysstime}) + \text{servtime} \\ r &= (n \times \text{sysstime}) + (n \times \text{servtime}) \end{aligned}$$

$$\frac{r}{m} = \frac{\text{sysstime} + \text{servtime}}{\text{sysstime} + \frac{\text{servtime}}{n}} = \frac{1 + \frac{\text{servtime}}{\text{sysstime}}}{1 + \frac{\text{servtime}}{\text{sysstime}} \times \frac{1}{n}}$$

$$\text{Let } T = \frac{\text{servtime}}{\text{sysstime}}$$

$$\text{Then } \frac{r}{m} = \frac{1 + T}{1 + \frac{T}{n}}$$

The above expression clearly shows that the speedup is sensitive to the value of *T*. In the limit, as *n* tends to

infinity, the value of r/m tends to $1 + T$. This accounts for the fact that the saturation value of the speedup is higher for longer server computation times. Because the times to send and receive a packet dominate *system* time, improvements to the underlying network primitives will improve the maximum speedup obtained with MultiRPC. Consequently, an improved basic RPC mechanism would result in improved MultiRPC performance rather than rendering it superfluous.

6.2. Experimental Results

We performed a series of carefully controlled experiments to confirm our understanding of MultiRPC and to explore its behaviour when contacting a large number of servers. We describe our experiments and the observations from them in the next three sections.

6.2.1. Experimental Environment

The experiments were conducted in an environment of about 500 Sun3, DEC MicroVax, and IBM RT-PC workstations running the Unix 4.2BSD operating system and attached to the Andrew File System. For uniformity, we ran our tests only on the IBM RT-PC workstations, with one of the workstations being the client and the others servers. By designing our tests to require no file accesses or system calls on the servers we avoided distortions of our measurements by distributed file system access.

The topology of the network connecting these workstations is shown in Appendix II. Although physically compact², there is considerable complexity in the network structure. There are about a dozen Ethernet and IBM Token Ring subnets connected to each other directly or via optic fibre links. Active computing elements, called *routers*, perform the appropriate forwarding or filtering of packets between these subnets. In addition to the Andrew workstations, many standalone workstations and mainframe computers are also on this network.

The scale and complexity of the network introduced serious problems in controlling our experiments. We had to be on guard against extraneous network activity loading the network and the routers. We also had to contend with the fact that closely-spaced replies to a MultiRPC call from a large number of servers could overload the routers and affect our measurements.

To address these problems we separated our experiments into two classes. The tests for model validation, discussed in Section 6.2.2, were run entirely on workstations located on a single subnet. For these tests we were able to ensure that there was no other network activity. The presence of routers was irrelevant since the client and all the servers were on the same subnet. Since there were only 20 workstations on this subnet, our model validation is restricted to this range.

The tests discussed in Section 6.2.3 to explore the large-scale performance of MultiRPC could not be controlled so well. To include more than 20 servers our tests had to span multiple subnets. We minimised the effects of extraneous network activity by running our experiments in the early hours of the morning, when the network was least active. Preliminary tests confirmed that this consistently produced smaller variances in our measurements than tests run at any other period of the day.

We simulated computation on the servers by delaying the reply to a request by a specified amount of time. Unfortunately, the clock resolution of 16 milliseconds on the RT-PCs was inadequate for the range of computation times of interest to us. We therefore had to resort to a timing loop to achieve delays of 10, 20

²The entire CMU campus is only about one square kilometer in size.

and 50 milliseconds in our tests. The clock resolution was also inadequate for measuring the elapsed time of individual calls at the client. To overcome this, we timed many iterations of each call and used the average.

6.2.2. Validation

The worst performance of MultiRPC, relative to RPC2, occurs when a single site is contacted. In this situation, the additional complexity of a MultiRPC call is not amortised over many connections. Table III-1 compares the elapsed times to contact a single site using RPC2 and MultiRPC, for zero server computation time. The table shows that the difference in times is negligible. Further, the RPC2 time presented in the table is no worse than the time observed on an earlier version of RPC2 that lacked MultiRPC support. Our design criterion of not slowing down simple RPCs has thus been met.

To compute the model predictions we need to know the times taken by individual components of MultiRPC. These values, obtained by standalone measurements of MultiRPC, are presented in Table III-2. By substituting these values in the expressions derived in Section 6.1 we obtain the model predictions shown in Table III-6 and Figure IV-4.

Figures IV-5, IV-6 and IV-7 compare the predictions of the model to our measurements. The model consistently predicts slightly better performance than we actually observe, but the fit is surprisingly good considering the simplicity of the model. For the reasons discussed earlier we were unable to investigate the validity of the model beyond 20 servers.

6.2.3. Large Scale Effects

Since the original motivation for MultiRPC involved a scenario with a large number of workstations distributed over the entire CMU campus, we were curious to see just how well MultiRPC behaves in such an environment. Our analytical model is not valid in this situation because of the presence of routers and extraneous network traffic. These factors affect both RPC2 and MultiRPC. Their effects show up as anomalies in the average values of the measured quantities, and as high associated variances.

Table III-3 and Figure IV-1 present our observations for server computation times of 10, 20 and 50 milliseconds. The behaviour below about 25 servers is in accordance with our model. Beyond this the speedup drops rather than increasing or remaining constant. Detailed examination of the data revealed an increase in the number of retried packets beyond about 25 servers. Below 25 servers, retries never comprised more than 0.4% of the total packets sent in a MultiRPC test. In fact, for those configurations there were often no retried packets at all. Beyond 25 servers the number of retried packets increased, and sometimes accounted for as much as 25 percent of the total traffic in configurations beyond 50 servers.

The measurements described above were performed with server computation times that were constants. We conjectured that one of the main reasons for poor behaviour at large scale was the overloading of routers due to simultaneous arrival of replies from many servers. Real server computations tend not to be constant. We therefore repeated the experiments with computation times normally distributed, with a standard deviation that was 10% of the mean. Table III-4 and Figure IV-2 show the corresponding results. We also repeated the experiments with an exponentially distributed service time, to provide validation data for a stochastic model that might be built in the future. This data is shown in Table III-5 and Figure IV-3. Unfortunately, as the results from these experiments show, the aberrations in performance at large scale do not vanish when using a non-constant service time distribution.

Although there is a decline in observed speedup in the large scale tests, it must be emphasised that MultiRPC performance is always better than iterative RPC2 performance. Further, we encountered no functional problems in using MultiRPC up to 100 servers. These facts give us confidence in the value of MultiRPC as a

basic component of large distributed systems.

7. Related Work

In this section we look at a number of parallel RPC mechanisms with a view to placing the design of MultiRPC in perspective. We make no attempt to be exhaustive in our examples or to be complete in our descriptions. Rather, our goal is to examine these systems in a manner that highlights their similarities and differences with respect to MultiRPC.

Work in the area of parallel network access has typically focused on broadcast or multicast protocols. Two examples of such work are the Sun Microsystems' Broadcast RPC (*Sun-BRPC*) [18], and Group Interprocess Communication in the V Kernel (*V-GIPC*) [3].

Sun-BRPC depends upon IP-level broadcast to communicate with multiple sites. Servers must register themselves in advance with a central port in order to be accessible via the broadcast facility. This is in contrast to MultiRPC where no explicit actions need be taken by the server; existing servers do not have to be modified, recompiled nor relinked. Like MultiRPC, Sun-BRPC provides for a client handler routine and an overall client-specified timeout. However it does not provide the same correctness guarantees and error reporting as MultiRPC.

V-GIPC uses the Ethernet multicast protocol as its basis and defines *host groups* as message addresses. Each request is multicast and it is up to each host to recognise those group addresses for which it has local processes as members. Reliable communication is not an objective of V-GIPC, even though its designers report that lost responses due to simultaneous arrival of packets are common. Another difference is that there is no notion of a client handler in V-GIPC. A client is blocked only until the first response is received; further responses have to be explicitly gathered. When another call is made, the previous call is implicitly terminated and further responses to it are discarded.

Neither Sun-BRPC nor V-GIPC allows long server computations at all sites in a parallel call while providing timely notification of site or network failures. A sufficiently long computation would simply cause a timeout. The MultiRPC retransmission protocol addresses this issue and allows the client to distinguish between a long computation and permanent communication failure.

There are also parallel RPC systems which do not depend on broadcast nor multicast. One such system is Circus [4, 5], which focuses on achieving high availability by using parallel RPC as a vehicle for replication. Circus is built on top of the DARPA IP/UDP protocol and is unique in that it supports many-to-many communication rather than one-to-many. It provides for a fixed set of routines called *collators*, one of which must be specified by the client when making a call. Collators perform a function similar to the MultiRPC client handler routine, but differ in that they do not allow a call to be terminated prematurely. Like MultiRPC, Circus uses probe packets to distinguish between long server computations and permanent communication failure. Many problems addressed by Circus, such as orphan detection and exactly-once semantics in the presence of failures, are unique to its intended application.

The Gemini parallel RPC mechanism [2, 11] is built on the reliable IP/TCP byte stream protocol [8], that subsumes the retransmission, timeout, acknowledgement and probe functions of RPC. It is similar to MultiRPC in that it requires no multicast or broadcast support. Unlike MultiRPC, a distinct Unix process is created on a server for each client. The stub compiler for Gemini accepts interface specification in C rather than defining a separate interface language. The equivalent of the MultiRPC client handler routine is a language construct called a *result statement*. This is a compound statement in C that syntactically appears

after a parallel remote procedure call. This body of code is executed exactly once for each reply and the execution can terminate the entire call prematurely.

8. Conclusion

The central message of this paper is that it is possible to build an efficient and easy to use parallel invocation mechanism whose semantics is a natural extension of the remote procedure call paradigm. We have derived an analytic model of this mechanism and shown that its predicted performance closely matches the measured performance of our implementation. Asymptotic analysis of this model indicates that improvements to the underlying transmission primitives would further strengthen the case for using MultiRPC in preference to iterative RPCs. We have demonstrated experimentally that the mechanism works successfully for up to 100 servers in a single call, executing in a complex network environment with diverse transmission media and interconnecting elements. Comparison with other parallel invocation mechanisms shows that MultiRPC is unique in its overall design, although some of the individual concepts used in it may be found elsewhere.

There are a number of ways in which the work reported here may be extended. First, MultiRPC provides parallelism at the programming interface but does not require multicast capability in any of the lower levels of the networking software. Suppose, however, multicast were available. How could MultiRPC use it? What would its performance behaviour be then? Our view is that multicast is a performance enhancement rather than a fundamental programming primitive. Preliminary work indicates that if the programmer is required to explicitly group connections, the semantics of MultiRPC can be preserved while using multicast internally. An interesting security question arises in this context. How does one support site-specific encryption on multicast packets? Our approach is to use the underlying secure RPC2 connections as key distribution channels for internally generated group-specific keys. We will report on our experience with this approach and further details on the use of multicast in a later paper.

Another potential improvement to MultiRPC involves ARGs. RP2Gen does not take advantage of repeated argument types when it generates ARGs; it creates a new ARG for each parameter of each remote operation. For recursive structure arguments this can consume a significant amount of storage. Since structure arguments are defined in the subsystem specification file, and hence known to RP2Gen, it should be possible to share ARGs.

Finally, using software in a variety of applications often results in unexpected lessons and refinements. It is impossible to predict in advance what such changes might be in the context of MultiRPC. However, we are confident that MultiRPC and mechanisms similar to it will prove to be an important building block in distributed systems.

Acknowledgements

We wish to thank Tom Holodnik of the Data Communications Group at Carnegie Mellon University for permission to use Figure II-1. We would also like to express our appreciation to Jay Kistler, Daniel Duchamp, Bradley White and Eric Cooper for their careful reading of this paper and their many useful comments.

I. An Example

This appendix presents a brief example in order to make some of the previous discussion more concrete. Although this example is contrived, it is adequate to illustrate the structure of a client and server that communicate via RPC2, and the changes that must be made to the client to use MultiRPC.

The subsystem designer defines a subsystem, chooses a name for it, and writes the specifications in the file *<subsystemname>.rpc2*. This file is submitted to RP2Gen, which generates client and server stubs and a header file. RP2Gen names its generated files using the file name of the *.rpc2* file with the appropriate suffix.

Figure I-1 presents the specification of the subsystem *example* in the file *example.rpc2*. RP2Gen interprets this specification and produces a client stub in the file *example.client.c*, a server stub in *example.server.c* and a header file *example.h* (shown in Figure I-2). This subsystem is composed of two operations, *double_it* and *triple_it*. These procedures both take a call by value-result parameter, *testval*, containing both the integer to be operated on and the result returned by the server.

Once the interface has been specified, the subsystem implementor is responsible for exporting the subsystem and writing the server main loop and the bodies of the procedures to perform the server operations. A client wishing to use this server must first bind to it and then perform an RPC on that connection. Figures I-3 and I-4 illustrate the client and server code. We wish to emphasise that this code is devoid of any considerations relating to MultiRPC.

Now consider extending this example to contact multiple servers using MultiRPC. The *example.rpc2* file and the server code remain exactly the same. Argument Descriptor Structures (ARGs), used by MultiRPC to marshal and unmarshal arguments, are already present in the client stub file; pointers to these structures are defined in the *example.h* file. Only the client code has to be modified, as shown in Figures I-5 and I-6.

From the client's perspective, a MultiRPC call is slightly different from a simple RPC2 call. The procedure invocation no longer has the syntax of a local procedure call. Instead, the single library routine *MAKEMULTI* is used to access the runtime system routine *MULTIRPC*. The client must allocate all necessary parameter storage. IN arguments are simply supplied as for any procedure call, but for OUT and IN-OUT parameters arguments arrays of pointers to the appropriate types must be supplied to the *MAKEMULTI* routine. The return arguments from each of the servers will be placed in the appropriate elements of the arrays.

The client is also responsible for supplying a handler routine for any server operation which is used in a MultiRPC call. The handler routine is called by MultiRPC as each individual server response arrives, providing an opportunity to perform incremental bookkeeping and analysis. The return code from the handler gives the client control over the continuation or termination of the MultiRPC call.

RP2Gen specification file for simple subsystem

```
Server Prefix "serv";                                tag server operations to avoid ambiguity

Subsystem "Example";

double_it(IN OUT RPC2_Integer) testval);
triple_it(IN OUT RPC2_Integer) testval);
```

Figure I-1: *example.rpc2* specification file

.h file produced by RP2GEN
Input file: example.rpc2

```
#include "rpc2.h"
#include "se.h"
```

Op codes and definitions

```
#define double_it_OP    1
extern ARG double_it_ARGS[ ];
#define double_it_PTR    double_it_ARGS

#define triple_it_OP    2
extern ARG triple_it_ARGS[ ];
#define triple_it_PTR    triple_it_ARGS
```

Figure I-2: The RP2Gen-generated header file *example.h*

Include relevant header files

```
main()
```

```
{
    int testval, op;
    RPC2_Handle cid;
```

Perform LWP and RPC2 Initialization

Establish connections to server using RPC2_Bind
 RPC2_Bind(Bind arguments, &cid);

```
while (TRUE)
```

```
{
    printf("\nDouble [ = 1] or Triple [ = 2] (type 0 to quit)? ");
    scanf("%d", &op);
    if (op == 0)
        break;
    printf("Number? ");
    scanf("%d", &testval);
    Perform RPC2 call on connection cid
    if (op == 1) double_it(cid, testval);
    else triple_it(cid, testval);
    printf("result = %d\n", testval);
}
```

```
RPC2_Unbind(cid);
printf("Bye...\n");
}
```

Terminate connection with server

Figure I-3: A Simple RPC2 Client

Include relevant header files

main()

```
{
  RPC2_PacketBuffer *reqbuffer;
  RPC2_Handle cid;
```

Perform LWP and RPC initialization

Set filter to accept requests on new or existing connections

Enter server loop

```
while(TRUE)
```

```
{
```

Await a client request:

```
  RPC2_GetRequest(&cid, &reqbuffer, Other Arguments)
```

```
  serv_ExcuteRequest(cid, reqbuffer) Routine is generated by RP2Gen
```

```
}
```

```
}
```

Bodies of server procedures

```
long serv_double_it(cid, testval)
```

```
  RPC2_Handle cid;
```

```
  RPC2_Integer *testval;
```

```
{
```

```
  *testval = (*testval) * 2;
```

```
  return(RPC2_SUCCESS);
```

```
}
```

```
long serv_triple_it(cid, testval)
```

```
  RPC2_Handle cid;
```

```
  RPC2_Integer *testval;
```

```
{
```

```
  *testval = (*testval) * 3;
```

```
  return(RPC2_SUCCESS);
```

```
}
```

Figure I-4: A Simple RPC2 Server

```

Include relevant header files
#define HOWMANY 3
#define WAITFOR 2
long HandleDouble();
long HandleResult();
long returns;

main()
{
    int testval[HOWMANY], op;           Can use static or dynamic allocation
    RPC2_Handle cid[HOWMANY];

    Perform LWP and RPC2 Initialization

    Establish connections to servers
    for(count = 0; count < HOWMANY; count++)
    {
        ret = RPC2_Bind(Bind arguments, &cid[count]);
        testval[count] = (int*)malloc(sizeof(int));    allocate space for arguments
    }

    while (TRUE)
    {
        printf("\nDouble [ = 1] or Triple [ = 2] (type 0 to quit)? ");
        scanf("%d", op);
        if (op == 0) break;
        printf("\nNumber? ");
        scanf("%d", *testval[0]);    IN argument goes in 1st array slot

        Make the MultiRPC call
        returns = 0;
        bzero(testval, HOWMANY*sizeof(int));    initialize results
        if (op == 1)
            MakeMulti(double_it_OP, double_it_PTR, HOWMANY, cid,
                HandleDouble, NULL, testval);
        else MakeMulti(triple_it_OP, triple_it_PTR, HOWMANY, cid,
            HandleTriple, NULL, testval);
        for (count = 0; count < HOWMANY; count++)
            printf("result[%d] = %d\n", count, testval[count]);
    }

    for (count = 0; count < HOWMANY; count++)
        RPC2_Unbind(cid[count]);
    printf("Bye...\n");
}

```

Figure I-5: Client using MultiRPC


```

long HandleDouble(HowMany, cidarray, host, rpcval, testval)
    RPC2_Integer HowMany, host, rpcval, *testval[];
    RPC2_Handle cidarray[];
    {
        if (rpcval != RPC2_SUCCESS)
            printf("HandleDouble: rpcval = %d\n", rpcval);
        if (++returns == WAITFOR) return -1;    Terminate the MultiRPC call
        return(0);                             Continue accepting server responses
    }

long HandleTriple(HowMany, cidarray, host, rpcval, testval)
    RPC2_Integer HowMany, host, rpcval, *testval[];
    RPC2_Handle cidarray[];
    {
        if (rpcval != RPC2_SUCCESS)
            printf("HandleTriple: rpcval = %d\n", rpcval);
        if (++returns == WAITFOR) return -1;    Terminate the MultiRPC call
        return(0);                             Continue accepting server responses
    }

```

Figure I-6: MultiRPC Client Handler Routines

II. Network Topology

Carnegie Mellon Internet

April 10, 1987

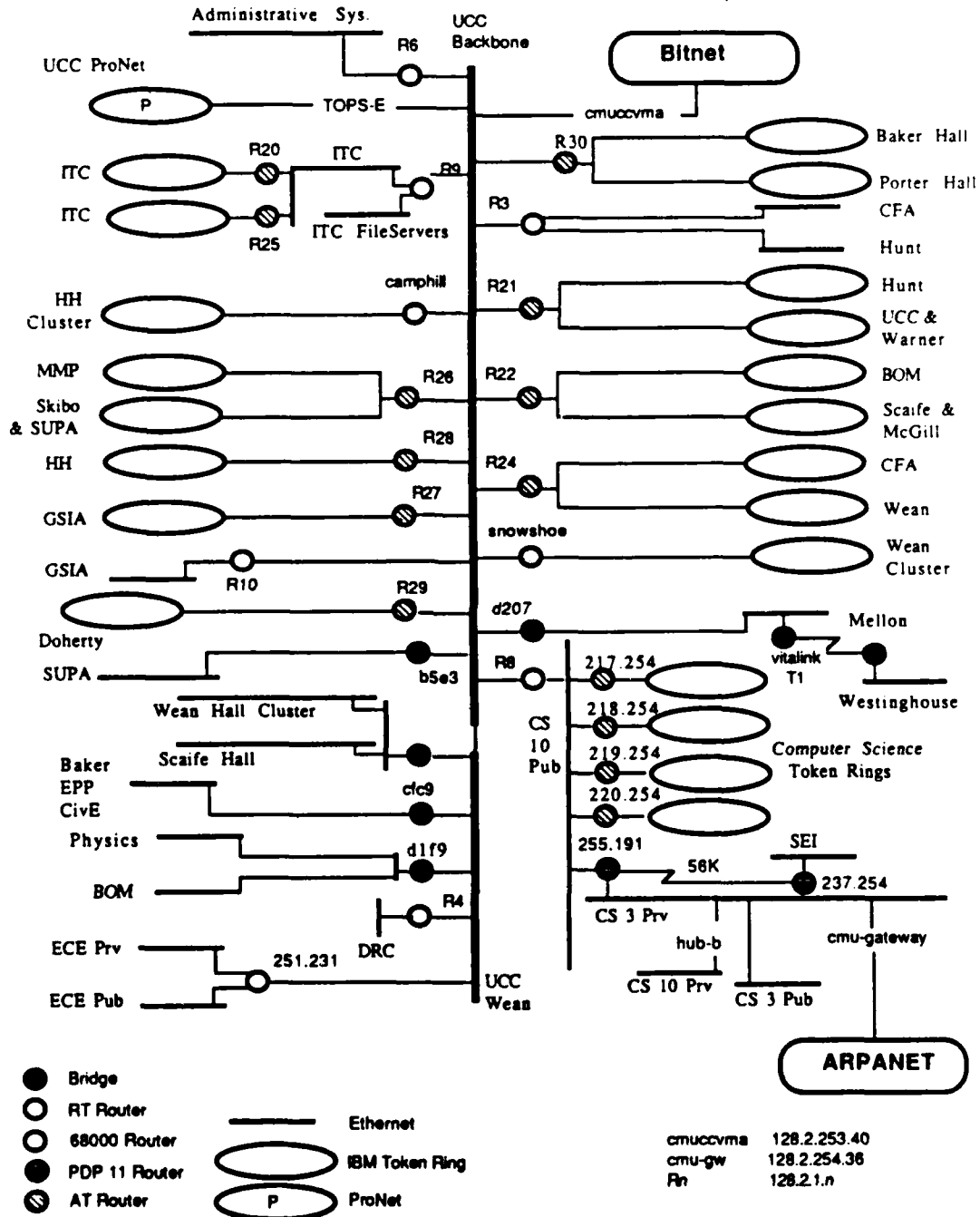


Figure II-1: Carnegie Mellon Network Topology

III. Tables

MRPC	RPC2	R / M
29.00 (0.97)	27.80 (1.01)	0.96

Figures in parenthesis are standard deviations.

Table III-1: Measured Times for null RPC2 and MultiRPC Calls (in ms)

<i>pack</i>	<i>clch</i>	<i>servoh</i>	<i>clproc</i>	<i>unpack</i>	<i>send</i>
0.56	5.98	13.69	3.93	0.20	5.15

Table III-2: Average Times for Individual Components of MultiRPC (in ms)

Servers	10 ms Computation			20 ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	39.30 (0.73)	38.30 (1.13)	0.97	49.70 (1.08)	47.30 (1.34)	0.95	81.10 (6.37)	78.80 (6.13)	0.97
2	45.90 (0.95)	76.50 (1.10)	1.67	56.40 (0.94)	96.10 (1.21)	1.70	85.60 (7.79)	158.35 (7.43)	1.85
5	66.25 (1.07)	191.55 (2.19)	2.89	76.45 (0.94)	242.40 (4.08)	3.17	104.35 (7.63)	391.20 (3.52)	3.75
7	79.80 (1.15)	268.70 (3.77)	3.37	89.90 (1.02)	336.25 (3.35)	3.74	120.95 (8.68)	549.75 (6.56)	4.55
10	107.90 (1.21)	385.10 (11.50)	3.57	110.55 (1.10)	482.05 (7.78)	4.36	139.50 (7.90)	784.80 (9.57)	5.63
13	140.10 (1.07)	500.80 (9.51)	3.57	139.95 (1.00)	627.35 (7.74)	4.48	161.40 (7.24)	1046.30 (37.60)	6.40
15	161.30 (1.08)	579.10 (12.82)	3.59	161.35 (1.23)	725.95 (8.20)	4.50	175.20 (9.00)	1179.25 (14.02)	6.73
17	182.40 (0.99)	654.10 (10.47)	3.59	183.35 (1.50)	820.05 (8.31)	4.47	188.75 (10.07)	1337.50 (12.52)	7.09
19	204.10 (1.07)	740.85 (22.37)	3.63	205.60 (2.52)	915.55 (55.51)	4.45	209.35 (13.62)	1496.95 (17.00)	7.15
50	766.10 (26.30)	2744.00 (73.36)	3.58	766.50 (20.28)	3130.90 (59.34)	4.08	796.90 (26.21)	4912.50 (255.90)	6.16
75	•	•	•	1225.20 (114.13)	4985.53 (86.52)	4.07	1331.20 (433.06)	7707.90 (245.59)	5.79
100	1705.90 (27.00)	6017.20 (222.09)	3.52	1780.40 (136.46)	6844.00 (246.76)	3.84	1429.50 (217.27)	9947.00 (212.63)	6.96

All times are in milliseconds. Figures in parentheses are standard deviations. A subset of this data is graphically presented Figure IV-1 and is also used in Figures IV-5, IV-6 and IV-7. For configurations up to 20 servers, all machines were located on a single isolated token ring. Configurations involving more than 20 servers spanned multiple network segments. Each data point was obtained from 10 trials.

Table III-3: Measured Call Times for Constant Server Computation Times

Servers	10 ms Computation			20 ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	39.32 (1.11)	37.75 (0.95)	1.02	50.40 (6.42)	47.60 (1.00)	0.94	77.96 (7.92)	78.40 (12.28)	1.01
2	44.92 (1.29)	75.92 (1.58)	1.69	59.56 (6.68)	99.88 (10.41)	1.68	85.28 (7.74)	155.96 (12.06)	1.83
5	70.16 (3.25)	208.52 (3.74)	2.97	79.32 (2.98)	261.68 (5.75)	3.30	108.36 (12.48)	414.24 (31.02)	3.82
10	112.55 (3.97)	478.10 (16.40)	4.25	112.08 (9.15)	530.00 (17.95)	4.73	143.40 (10.01)	824.76 (19.72)	5.75
15	162.75 (18.40)	747.75 (21.09)	4.59	148.16 (2.19)	783.36 (20.76)	5.29	185.55 (14.50)	1270.85 (30.83)	6.85
19	•	•	•	•	•	•	224.00 (25.12)	1698.70 (42.18)	7.58
20	208.05 (7.10)	956.55 (35.47)	4.60	197.44 (2.16)	1065.76 (21.33)	5.40	•	•	•
25	247.48 (1.19)	1153.52 (18.49)	4.66	248.96 (5.14)	1430.00 (93.75)	5.74	258.04 (13.50)	2179.00 (47.62)	8.44
50	780.00 (34.57)	2933.10 (124.90)	3.76	771.79 (8.56)	3266.20 (56.84)	4.23	838.20 (34.70)	4901.70 (186.16)	5.85
75	1474.30 (103.62)	4579.60 (57.35)	3.12	1238.20 (177.80)	5351.07 (239.53)	4.32	1581.50 (429.48)	7690.20 (277.17)	4.86
100	1715.89 (28.35)	6120.10 (61.95)	3.57	1637.30 (121.99)	7282.70 (403.44)	4.45	1943.60 (1965.90)	10082.40 (248.46)	5.19

All times are in milliseconds. Figures in parentheses are standard deviations. This data was obtained from servers distributed over many network segments. A subset of this data is graphically presented in Figure IV-2. Each data point was obtained from 10 trials.

Table III-4: Measured Call Times with Normally Distributed Server Computation Times

Servers	10 ms Computation			20 ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	37.68 (3.98)	36.80 (6.32)	0.98	44.96 (5.14)	44.64 (5.42)	0.99	91.48 (55.27)	85.20 (51.35)	0.93
2	45.63 (4.14)	72.80 (5.35)	1.53	62.84 (7.71)	91.96 (8.97)	1.46	111.44 (63.78)	159.96 (73.82)	1.43
5	72.92 (4.15)	206.48 (14.25)	2.83	94.12 (7.01)	250.64 (27.30)	2.66	150.92 (46.55)	395.52 (120.55)	2.62
10	111.20 (2.17)	439.65 (22.78)	3.95	123.96 (7.71)	521.96 (80.40)	4.21	227.96 (92.85)	788.36 (115.91)	3.46
15	162.90 (1.52)	626.70 (53.22)	3.85	169.20 (12.97)	906.55 (82.43)	5.36	249.80 (60.33)	1203.56 (237.48)	4.82
19	211.95 (12.15)	843.95 (54.43)	3.98	•	•	•	•	•	•
20				210.05 (13.55)	1182.65 (108.17)	5.63	288.90 (72.33)	1654.80 (232.17)	5.73
25	•	•	•	256.60 (20.49)	1381.30 (88.92)	5.38	304.92 (78.89)	2175.48 (307.59)	7.13
50	768.20 (10.75)	2835.90 (170.28)	3.69	787.05 (14.77)	3166.40 (146.35)	4.02	912.60 (110.29)	4559.30 (282.20)	5.61
75	1358.11 (139.52)	4390.60 (94.70)	3.23	1227.73 (148.51)	5272.80 (462.43)	4.29	1015.63 (38.86)	7543.70 (521.72)	7.43
100	1675.63 (39.54)	6074.90 (118.74)	3.62	1586.40 (52.27)	7335.90 (419.85)	4.62	1518.90 (212.51)	10724.90 (2397.54)	7.06

All times are in milliseconds. Figures in parentheses are standard deviations. This data was obtained from servers distributed over many network segments. A subset of this data is graphically presented in Figure IV-3. Each data point was obtained from 10 trials.

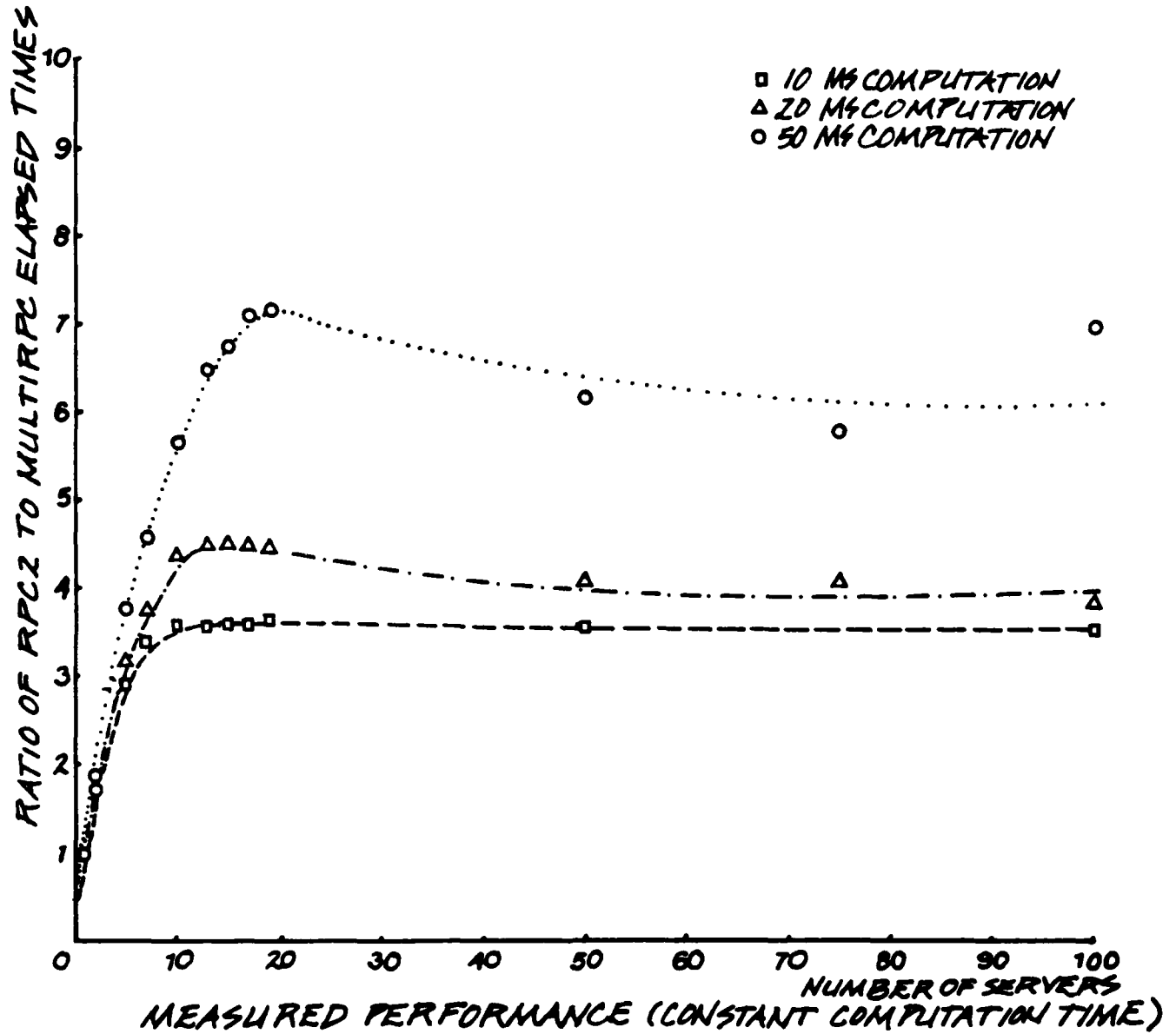
Table III-5: Measured Call Times with Exponentially Distributed Server Computation Times

Servers	10 ms Computation			20 ms Computation			50 ms Computation		
	MRPC	RPC2	R/M	MRPC	RPC2	R/M	MRPC	RPC2	R/M
1	34.36	37.80	1.10	44.36	47.80	1.08	74.36	77.80	1.05
2	39.32	75.60	1.92	49.32	95.60	1.94	79.32	155.60	1.96
5	54.20	189.00	3.49	64.20	239.00	3.72	94.20	389.00	4.13
7	71.33	264.60	3.71	74.12	334.60	4.51	104.12	544.60	5.23
10	101.66	378.00	3.72	101.66	478.00	4.70	119.00	778.00	6.54
13	131.99	491.40	3.72	131.99	621.40	4.71	133.88	1011.40	7.55
15	152.21	567.00	3.73	152.21	717.00	4.71	152.21	1167.00	7.67
17	172.43	642.60	3.73	172.43	812.60	4.71	172.43	1322.60	7.67
19	192.65	718.20	3.73	192.65	908.20	4.71	192.65	1478.20	7.67
20	202.76	756.00	3.73	202.76	956.00	4.71	202.76	1556.00	7.67
25	253.31	945.00	3.73	253.31	1195.00	4.72	253.31	1945.00	7.68
50	506.06	1890.00	3.73	506.06	2390.00	4.72	506.06	3890.00	7.69
75	758.81	2835.00	3.74	758.81	3585.00	4.72	758.81	5835.00	7.69
100	1011.56	3780.00	3.74	1011.56	4780.00	4.73	1011.56	7780.00	7.69

All times are in milliseconds. The server computation time is assumed constant. This data is graphically presented in Figure IV-4, and used in Figures IV-5, IV-6 and IV-7.

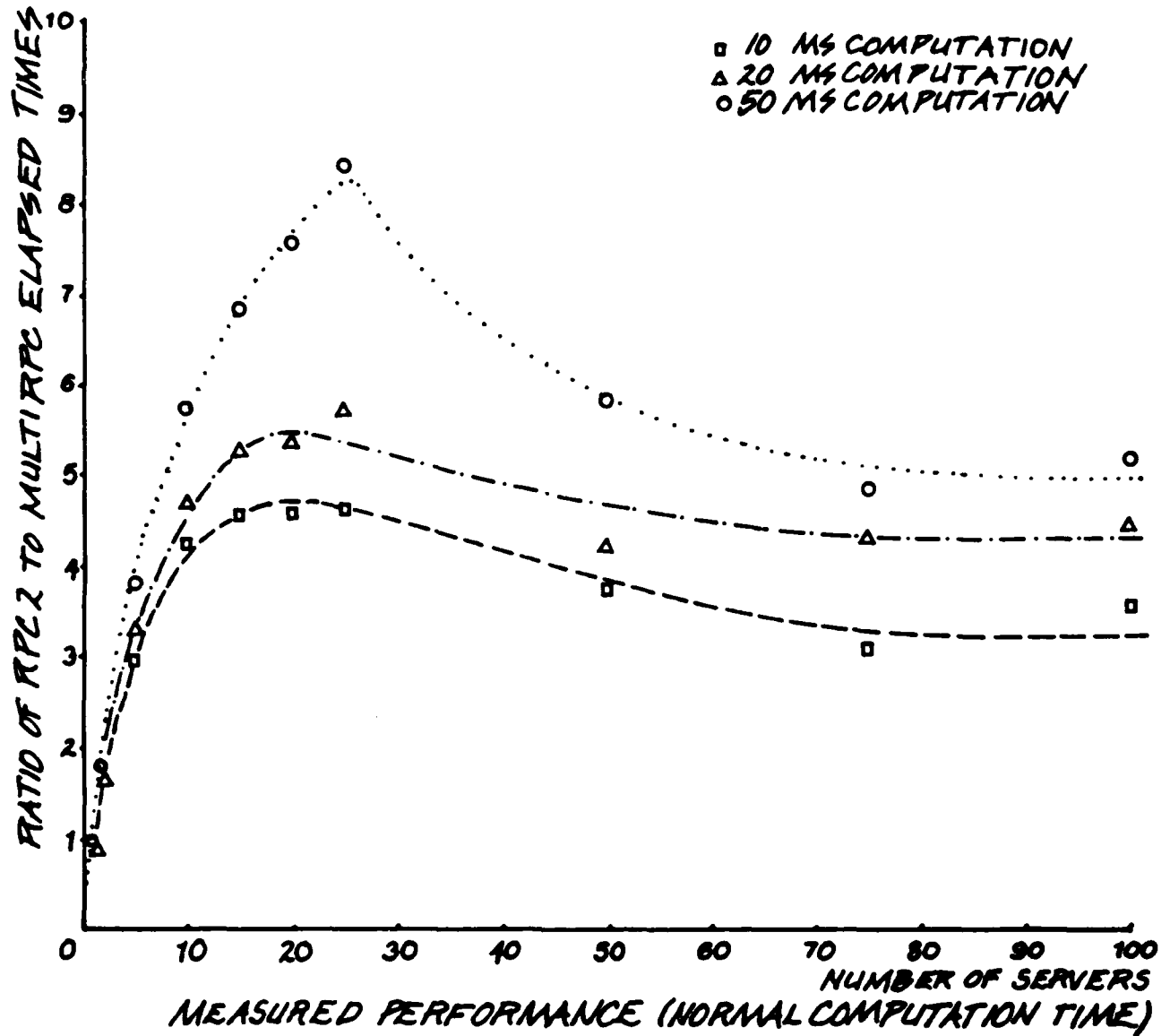
Table III-6: Predicted Performance from Analytic Model

IV. Graphs



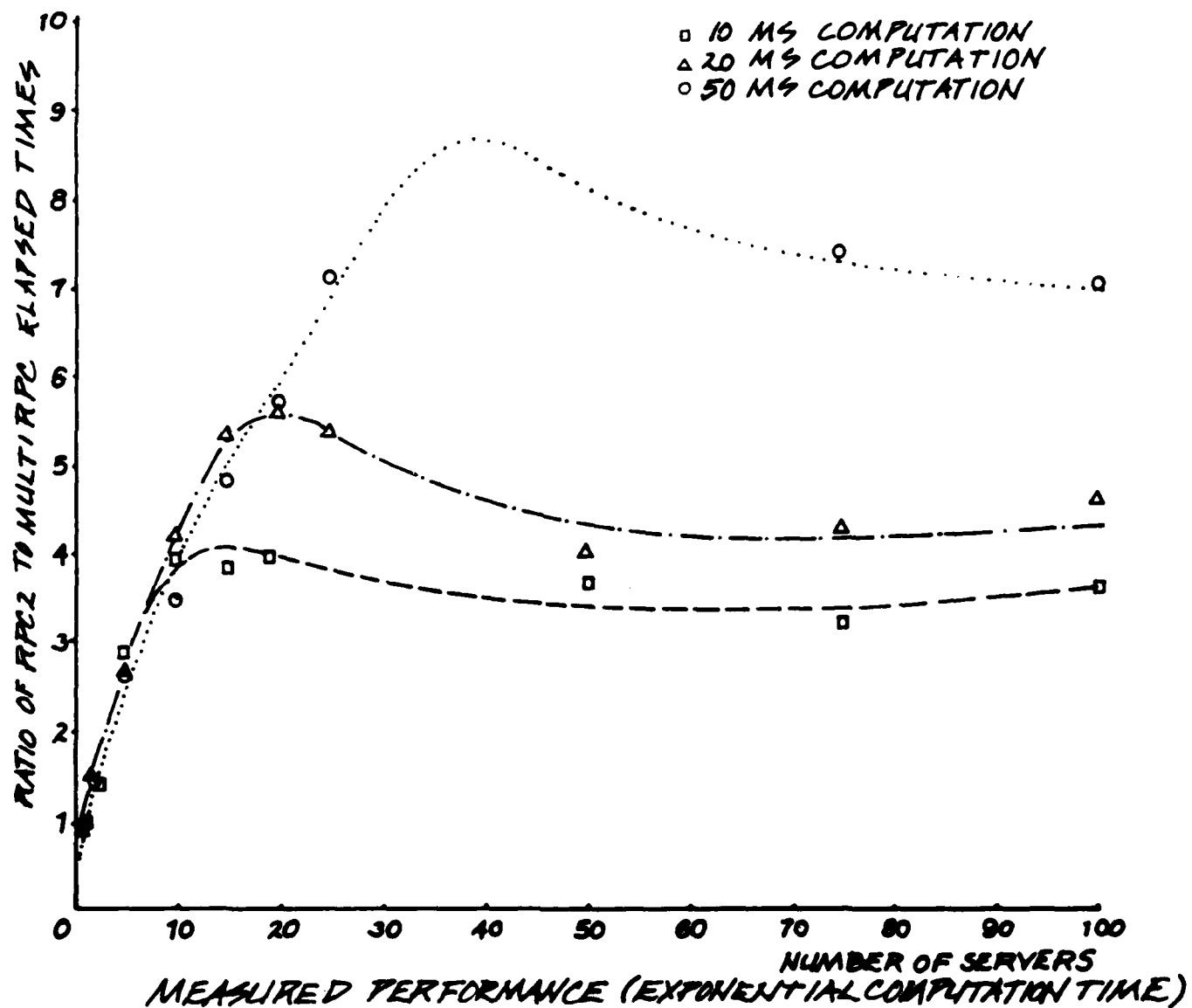
The data in this graph is obtained from Table III-3.

Figure IV-1: Measured Performance for Constant Server Computation Time



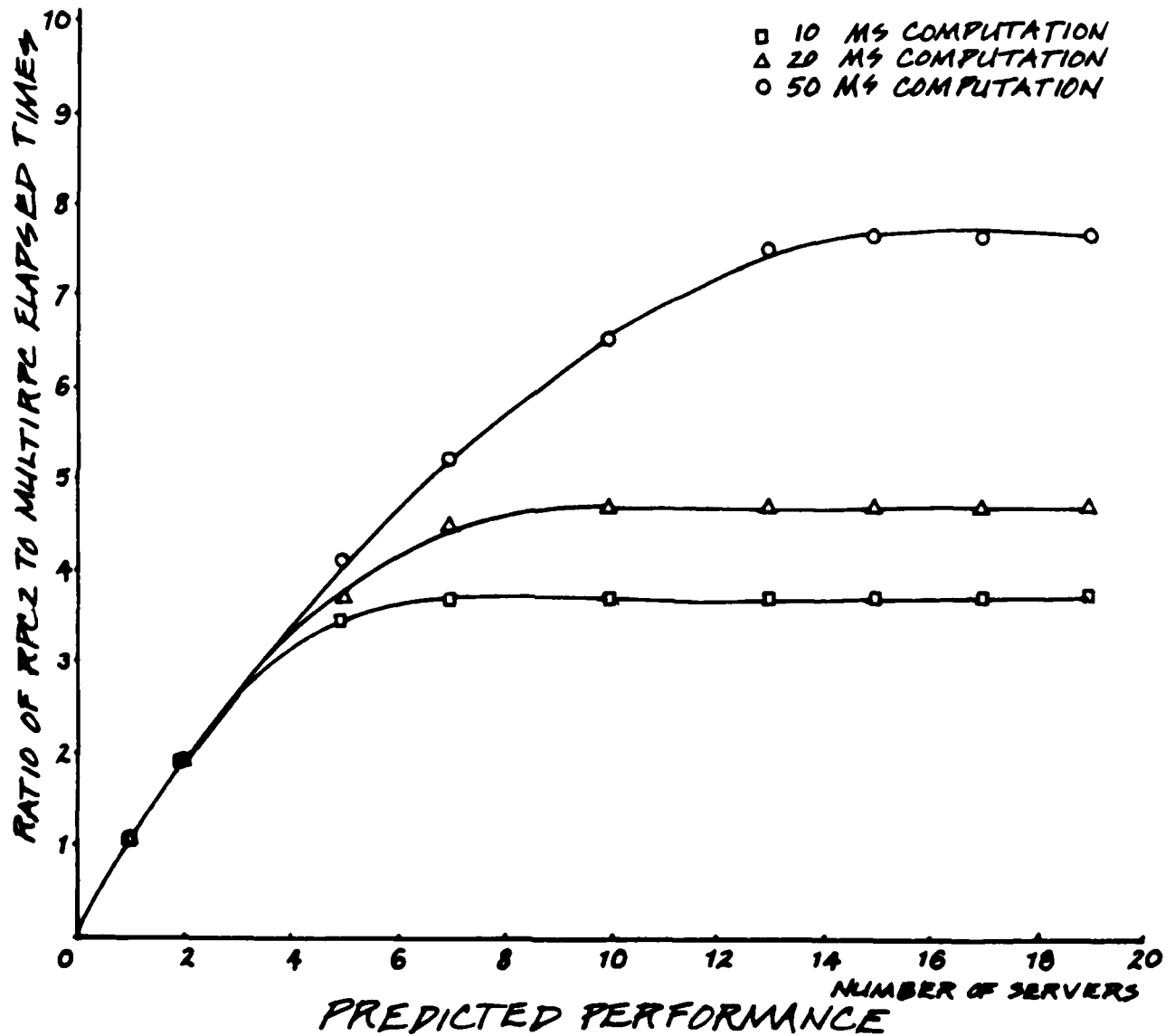
The data in this graph is obtained from Table III-4.

Figure IV-2: Measured Performance for Normal Server Computation Time



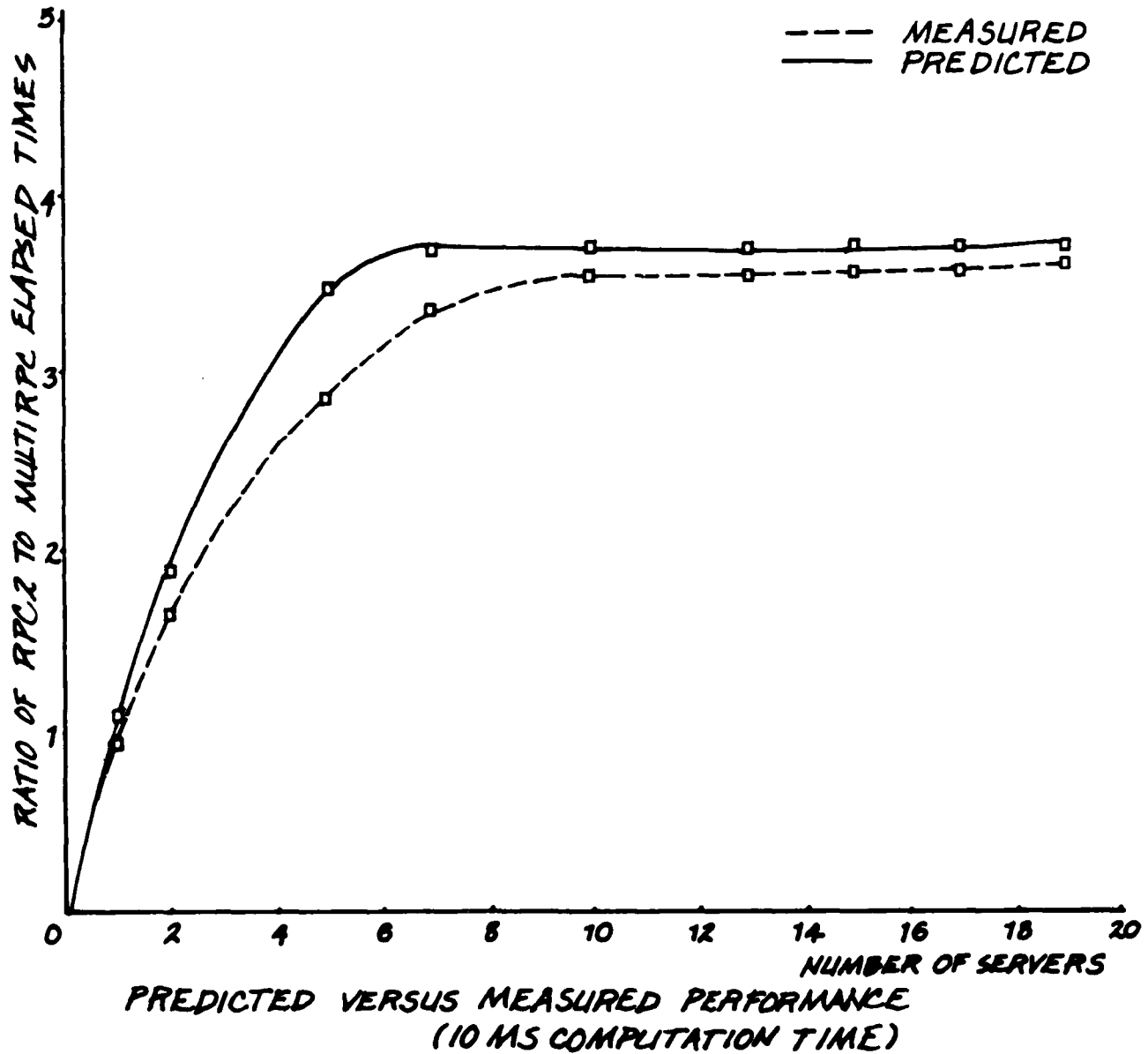
The data in this graph is obtained from Table III-5.

Figure IV-3: Measured Performance for Exponential Server Computation Time



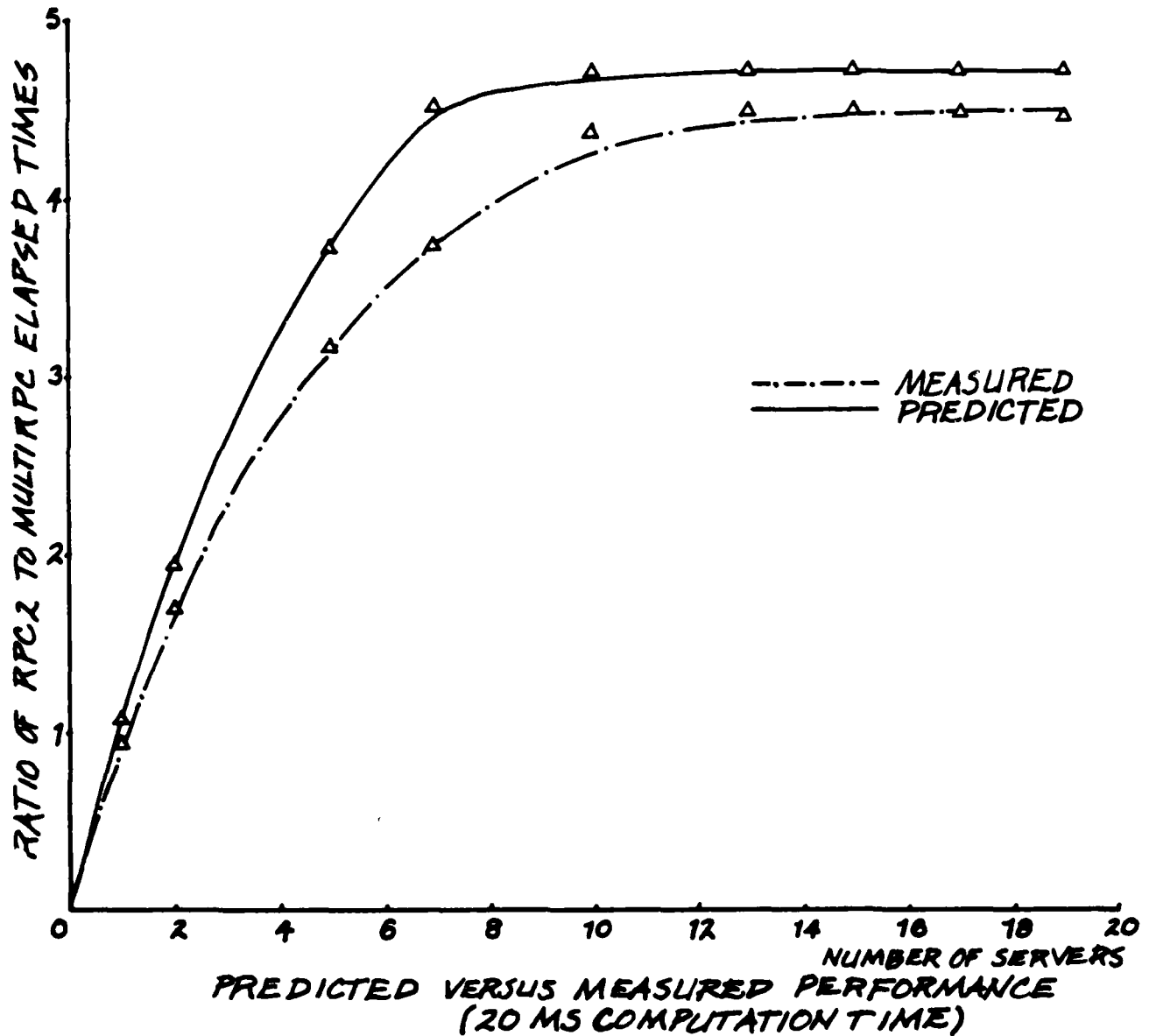
The data in this graph is obtained from Table III-6.

Figure IV-4: Predicted Performance from Model (Constant Computation Time)



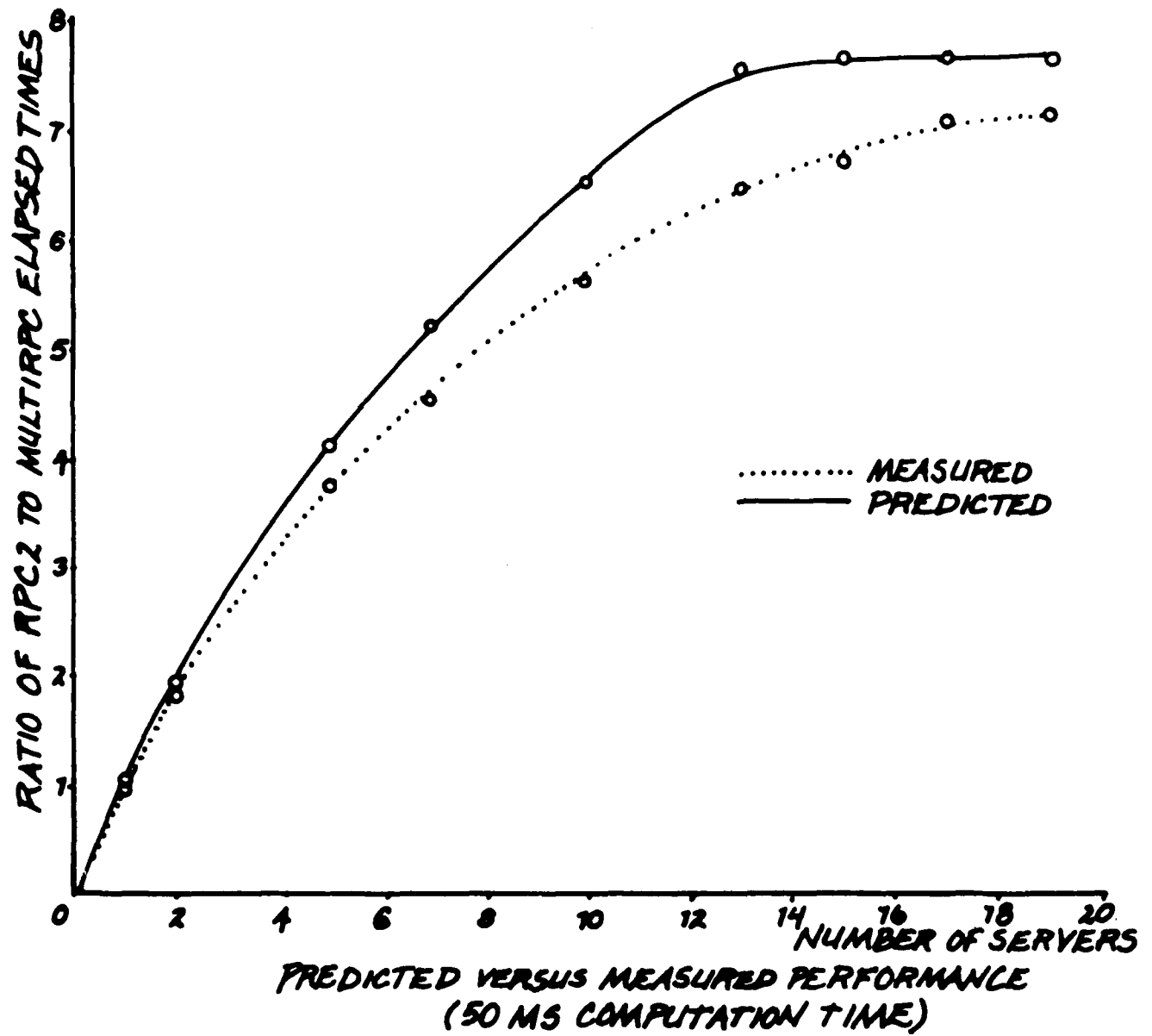
The predicted data is from Table III-6. The measured data is from Table III-3.

Figure IV-5: Comparison of Predicted and Measured Performance (10 ms Computation)



The predicted data is from Table III-6. The measured data is from Table III-3.

Figure IV-6: Comparison of Predicted and Measured Performance (20 ms Computation)



The predicted data is from Table III-6. The measured data is from Table III-3.

Figure IV-7: Comparison of Predicted and Measured Performance (50 ms Computation)

References

- [1] Birrell, A.D. and Nelson, B.J.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 1(1):39-59, February, 1984.
- [2] Burkhard, Walter A., Martin, Bruce E. and Paris, Jehan-Francois.
The Gemini Replicated File System Test-bed.
In *Proceedings of the Third International Conference on Data Engineering (forthcoming)*. 1987.
- [3] Cheriton, David R., and Zwaenepoel, Willy.
Distributed Process Groups in the V Kernel.
ACM Transactions on Computer Systems 2(2):77-107, May, 1985.
- [4] Eric C. Cooper.
Circus: A Replicated Procedure Call Facility.
Technical Report UCB/CSD 85/196, PROGRES Report No. 84.12, Computer Science Division,
University of California, January, 1985.
- [5] Cooper, Eric C.
Replicated Distributed Programs.
In *Proceedings of the Tenth ACM Symposium on Operating System Principles*. December, 1985.
- [6] Defense Advanced Research Projects Agency, Information Processing Techniques Office.
RFC 791: Internet Program Protocol Specification
September 1981.
- [7] Defense Advanced Research Projects Agency, Information Processing Techniques Office.
RFC 768: User Datagram Protocol Specification
September 1981.
- [8] J. Postel, ed.
RFC 793: Transmission Control Protocol - DARPA Internet Program Protocol Specification
September 1981.
- [9] Herlihy, M.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1):32-53, February, 1986.
- [10] Jones, M.B., Rashid, R.F., and Thompson, M.
MatchMaker: An interprocess specification language.
In *Proceedings of the ACM Conference on Principles of Programming Languages*. January, 1985.
- [11] Bruce Martin.
Parallel Remote Procedure Call Language Reference and User's Guide
Computer Systems Research Group, University of California, San Diego, 1986.
- [12] Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H., and Smith, F.D.
Andrew: A Distributed Personal Computing Environment.
Communications of the ACM 29(3):184-201, March, 1986.
- [13] Jonathan Rosenberg, Larry Raper, David Nichols, M. Satyanarayanan.
LWP Manual
Information Technology Center, CMU-ITC-037, 1985.

- [14] Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z. and West, M.J.
The ITC Distributed File System: Principles and Design.
In *Proceedings of the Tenth ACM Symposium on Operating System Principles*. December, 1985.
- [15] M.Satyanarayanan.
RPC2 User Manual
Information Technology Center, CMU-ITC-038, 1986.
- [16] M. Satyanarayanan.
RPC2: A Communication Mechanism for Large-Scale Distributed Systems.
Manuscript in preparation, 1986.
- [17] Spector, A.Z.
Performing remote operations efficiently on a local computer network.
Communications of the ACM 25(4):246-260, April, 1982.
- [18] Sun Microsystems Inc.
Networking on the Sun Workstation
15 May 1985.

END

DATE

FILMD

3-88

DTIC